Recap from last Python lecture

Interpreted, imperative, OO Language
• Everything is an object
• Dynamic Typing

Programs are made up of:
• Expressions
• Statements
  - Assignment
  - if/elif/else
  - while-loops
  - Functions
• Classes (still to come)
Today: Revisit some objects

- Exploit features and build powerful expressions

**Base**: `int`, `float`, `complex`

**Sequence**: `string`, `tuple`, `list`
What can sequences do?

Select

- i-th element: \( s[i] \)
- subsequence ("slice"): \( s[i:j] \)

Update -- For mutable sequences (e.g. Lists)

- Update i-th element: \( s[i] = e \)
- Update subsequence: \( s[i:j] = e \)
Update subsequence: $s[i:j] = e$

- Changes the “object” referred to by $s$
- May change the length of the sequence
  - Increase: if RHS length $> j - i$
  - Decrease: if RHS length $< j - i$
Update subsequence

>>> z = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> z[3:6] = ["a", "b", "c"]
>>> z
[1, 2, 3, "a", "b", "c", 7, 8, 9, 10]
>>> z[3:6] = ["a", "b"] * 2
>>> z
[1, 2, 3, "a", "b", "a", "b", 7, 8, 9, 10]
>>> z[4:] = []
>>> z
[1, 2, 3, "a"]
>>> z[:0] = ["al", "be"]
>>> z
["al", "be", 1, 2, 3, "a", "b", "a", "b", 7, 8, 9, 10]
What else can sequences do?

Q: Suppose you are given a sequence $s$
How to find if the element $x$ appears in $s$?

$x \text{ in } s$

Works for any sequence type ...
Sequence “contains”  $x \text{ in } s$

```python
>>> "a" in "cat"
True
>>> "a" in "entebbe"
False
>>> "a" in ("c", "a", "t")
True
>>> 2 in [1,2,3,4,5]
True
>>> 2 in [1,4,92,2.4]
False
```
What can sequences do?

Select
- i-th element: \(s[i]\)
- subsequence ("slice"): \(s[i:j]\)

Update -- For mutable sequences (e.g. Lists)
- Update i-th element: \(s[i] = e\)
- Update subsequence: \(s[i:j] = e\)

Member
- Is an element in a sequence: \(x \text{ in } s\)
Doesn’t Python have For-Loops?

Why haven’t we seen For-loops yet?

- Because they are connected to sequences

For-loops are used to iterate over sequences

- Unlike in C, but similar to new Java foreach
- Elegant, powerful mechanism - use it!

```python
def body(x):
    for x in s:
        <BODY>
```

```python
def body(s):
    x = s[0]
    <BODY>
    x = s[1]
    <BODY>
    ...
    x = s[len(s) - 1]
    <BODY>
Iteration

```python
>>> for x in ["Midterms", "ain't", "cool"]:
    print x, len(x)

Midterms  5
ain't  5
cool  4
```

Works for any sequence ...

```python
>>> for c in "chimichanga":
    print c*3

ccc
hhh
iii
mmm ...
```
Iteration

```python
>>> s=0
>>> z=(1,2,3,4.0,"5")  #tuple
>>> for i in z:
    s = s + i
ERROR
>>> s
10
```

Can’t add string to float

- Note that first 4 elts added!
- Dynamic Types!
- Run-time Type Error
Iteration + binding for x,... in s:

If s is a sequence of tuples/sequences, then we can Bind to individual elements of “subsequences”

```python
>>> craigslist = [(
    "alien",3.50),
    ("dinosaur",1.90), ("quiz",100.50),
    ("quesadilla",3.00), ("good grade in 130","priceless")]

>>> for i,p in craigslist:
    print "One",i,"costs",p
One alien costs 3.5
One dinosaur costs 1.9
One quiz costs 100.5
One quesadilla costs 3.0
One good grade in 130 costs priceless
```
Old school For-loops

There’s a simple way to write good-old for-loops

Built-in function: \texttt{range}

\begin{verbatim}
for(i=0,i<10,i++){
    print i;
}
\end{verbatim}

\begin{verbatim}
>>> range(10)
[0,1,2,3,4,5,6,7,8,9]

>>> for i in range(10):
    print i

>>> range(5,15)    #fixed upper bound
[5,6,7,8,9,10,11,12,13,14]

>>> range(15,5,-1) #step
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6]
\end{verbatim}
But lookout!

For-loops are used to iterate over sequences

```python
for x in s:
    <BODY>
```

What if object referred to by \textit{s} is changed in \textit{BODY}?

Unpleasantness ensues:
- Try to ensure this never happens
- Iterate over a “copy” of the object
  ```
  - s[:]
  ```
Def funny_fun(s):
    for x in s:
        print x
    s[len(s):] = [x]

Adds x to end object being iterated over!
- Loops forever

Def dup_by_k(s,k):
    for x in s:
        print x
        s = s + x*k
return s

Creates new object w/ x*k added at end

Iteration object is what s “originally” referred to, which is unchanged
But lookout!

**def funny_fun(s):**

```python
for x in s:
    print x
s[len(s):] = [x]
```

Adds `x` to end object being iterated over!

- **Loops forever**

**def dup_by_k(s,k):**

```python
for x in s[:]
    print x
s = s + x*k
return s
```

Creates new object w/ `x*k` added at end

To make it more readable

Iteration object is what `s` "originally" referred to, which is unchanged
What can sequences do?

Select
- **i-th element**: `s[i]`
- **subsequence (“slice”)**: `s[i:j]`

Update -- For **mutable** sequences (e.g. Lists)
- **Update i-th element**: `s[i] = e`
- **Update subsequence**: `s[i:j] = e`

Member: `x in s`

Iteration: `for x in s: <body>`
What else?

Three useful functions for lists from ML?

- map
- filter
- fold (a.k.a. reduce)

Built-in in Python:
map

```python
def dup(x):
    return 2*x

>>> z = range(10)
>>> z
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> map(dup, z)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
>>> map(dup, "chimichanga")
["cc", "hh", "ii", "mm", "ii", "cc", "hh", "aa", "nn", "gg", "aa"]
```

- Works for all sequences, returns a list
- More flexible ways to call it, see documentation
filter

- Works for all sequences, returns same kind of sequence

```python
>>> def even(x): return int(x)%2==0
>>> filter(even,range(10))
[0,2,4,6,8]
>>> filter(even,"1234096001234125")
"240600242"
>>> filter(even,(1,2.0,3.2,4))
(2,4)
```

- Again, note the polymorphism that we get from dynamic types and conversion
reduce

• i.e. fold

```python
>>> def add(x, y): x+y
>>> reduce(add, range(10), 0)
45
>>> def fac(x):
    def mul(x, y): return x*y
    return reduce(mul, range(1, x+1), 1)
>>> fac(5)
120
```
What can sequences do?

Select

- **i-th element**: $s[i]$
- **subsequence (“slice”)**: $s[i:j]$

Update -- For **mutable** sequences (e.g. Lists)

- Update **i-th element**: $s[i] = e$
- Update **subsequence**: $s[i:j] = e$

Member: $x \text{ in } s$

Iteration: for $x \text{ in } s$: <body>
map, filter, reduce
List Comprehensions

A cleaner, nicer way to do map-like operations

```python
>>> [ x*x for x in range(10) ]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [ 2*x for x in "yogurt cheese" ]
["yy", "oo", "gg", "uu", "rr", "tt", ...]
```
List Comprehensions

Syntax: $>>> [e_x \text{ for } x \text{ in } s]$ 

Equivalent to:
List Comprehensions

Syntax: \[ e_x \text{ for } x \text{ in } s \]

Equivalent to:

```python
>>> def map_fn(x): return e_x
>>> map(map_fn, s)
```
List Comprehensions

A cleaner, nicer way to do map+filter-like operations

>>> [ x*x for x in range(10) if even(x) ]
[0, 4, 16, 36, 64]
>>> [ 2*x for x in "0123456" if even(x) ]
["00", "22", "44", "66"]
>>> [z[0] for z in craigslist if z[1]<3.0]
["dinosaur"]
List Comprehensions

Syntax:  

```python
>>> [e_x for x in s if c_x]
```

Equivalent to:
List Comprehensions

Syntax:  >>> \[e_x \text{ for } x \text{ in } s \text{ if } c_x \]  

Equivalent to:  

>>> def map_fn(x): return e_x
>>> def filter_fn(x): return c_x
>>> map(map_fn, filter(filter_fn, s))
List Comprehensions

Can “nest” the for to iterate over multiple sequences

```python
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

```python
>>> [(x, y) for x in range(3) for y in range(3) if x > y]
[(1, 0), (2, 0), (2, 1)]
```
What can sequences do?

Select
- i-th element: \( s[i] \)
- subsequence ("slice"): \( s[i:j] \)

Update -- For **mutable** sequences (e.g. Lists)
- Update i-th element: \( s[i] = e \)
- Update subsequence: \( s[i:j] = e \)

Member: \( x \ in \ s \)

Iteration: \( \text{for } x \ in \ s: \ <\text{body}> \)
map, filter, reduce

Comprehensions: \[ e_x \ for \ x \ in \ s \ if \ c_x \]
def sort(L):
    if L == []: return L
    else:
        l = sort(...)
        r = sort(...)
        return(l+L[0:1]+r)
def sort(L):
    if L==[]: return L
    else:
        l=sort([x for x in L[1:] if x < L[0]])
        r=sort([x for x in L[1:] if x >= L[0]])
        return(l+L[0:1]+r)
Today: Revisit some objects

- Exploit features and build powerful expressions

**Base:** `int`, `float`, `complex`

**Sequence:** `string`, `tuple`, `list`

**Maps (Dictionary):** `key` → `value`
Key data structure: Dictionaries

Associative arrays, Hash tables ...

A table storing a set of “keys”,
And a “value” for each key.

Any (immutable) object can be a key!
• int, float, string, tuples...

Very useful!
Using Dictionaries

Unsorted list of key, value pairs

Empty Dictionary: `{ }`

Non-empty Dictionary: `{k1:v1, k2:v2, ...}`

Membership: is k in dict: `k in d`

Lookup value of key: `d[k]`

Set value of key: `d[k]=v`
Dictionaries

```
>>> d={}
>>> d=dict(mexmenu)
>>> d[“ceviche”] = 3.95
>>> d
d
{…}
>>> d[“burrito”]
3.50
>>> d.keys()
...
>>> d.values()
```
def freq(s):
    d = {}
    for c in s:
        if c in d:
            d[c] += 1
        else:
            d[c] = 1
    return d

def plotfreq(s):
    d = freq(s)
    for k in d.keys():
        print k, "*"*d[k]

>>> d = plotfreq([1, 1, 3.0, "A", 3.0, "A", "A", 1, 2, 3.0, 1, "A"])
>>> d
...

>>> d = plotfreq("avrakedavra")
>>> d.keys()
>>> d
...
You now know enough to do PA5

- Python Tutorial: How to open files, read lines
- Use the `help` command
- Document every function: What does it do?